9. Покрокова розробка програм

Метод покрокової розробки програм використовують з метою спрощення роботи з великими програмами: для їхнього створення, перевірки правильності, модифікації.

Запровадження цього методу зумовлене обмеженими розумовими здібностями людини: неможливо, наприклад, охопити програму, що складається з кількох тисяч рядків. Дуже складно, або і неможливо, одразу почати писати програму для нової незнайомої задачі. Завжди доводиться виконувати певну підготовчу роботу, аналізуючи умову задачі та можливості комп'ютера, обмірковувати можливі варіанти майбутнього алгоритму.

Ідея методу покрокової розробки полягає у побудові послідовності програм, розрахованих на виконавців різної кваліфікації: від уявних, «надзвичайно розумних» спочатку до звичайного компілятора наприкінці. Початкова програма складається з кількох «абстрактних» операторів, виконати які в змозі лише наш уявний виконавець. Перехід до нової програми уточнює, деталізує ці оператори, «розшифровує» їх за допомогою дрібніших, ближчих до мови програмування. Деталізація закінчується, коли чергова програма цілковито записана операторами мови. Зазвичай метод покрокової розробки алгоритмів застосовують разом з дотриманням принципів структурного програмування і називають його *структурним програмуванням «зверху-донизу»*.

Нагадаємо, що структурована програма використовує лише ті структури керування, які мають один вхід і один вихід: послідовні дії, галуження, цикли. Якщо на кожному етапі розробки програми для деталізації абстрактних операторів використовувати тільки такі структури, то вона буде наділена вбудованою крок за кроком правильністю та надійністю.

Деталізацію передусім виконують для тих частин програми, які не залежать від інших частин і не зазнаватимуть ніяких змін надалі. Проілюструємо тепер все сказане на прикладі обчислення великого степеня числа 2 – такого, що результат неможливо зобразити за допомогою числа типу *int* чи, навіть, *long long int* (наприклад, 2^{1000}).

9.1. Обчислення великого степеня цілого числа

Задача 41. *Задано натуральне число n (n ≤ 1000). Обчислити і надрукувати 2^n.*

На найвищому рівні абстракції маємо таку програму:

```
void power0()
{
    надрукувати_заданий_степінь_2;
}
```

Програма *power0* настільки абстрактна, що може хіба ствердити наше бажання обчислити степінь двійки. Необхідно деталізувати її. Поміркуємо: завжди потрібно вводити вхідні дані задачі і завжди потрібно використовувати певні структури даних для відображення результатів. Тому отримаємо конкретнішу програму:

```
void power1()
{
    int n;
    cout << "Введіть n: ";
    cin >> n;
    велике_число x;
    обчислити_2^n_в_змінній_x;
    надрукувати_велике_число(x);
}
```

Введення вхідних даних є очевидним, тому ми записали його вже операторами мови C++. Деталізовано спосіб отримання результату: обчислення та виведення виконується різними операторами програми *power1*. Ці дії відокремлені одна від одної і далі можна їх уточнювати незалежно. Під час деталізації ми дотримувались принципів структурного програмування: абстрактний оператор з *power0* замінено *послідовністю* операторів у *power1*.

Звернемо увагу на одну важливу обставину: деталізація, з одного боку, уточнює і пояснює великі оператори, а з іншого – звужує клас можливих розв'язків. Наприклад, *power1* вже відхилила всі програми, які обчислюють і друкують результат по одній цифрі.

Що деталізувати далі? Рішення треба приймати для тих операторів, які не залежать від ще не прийнятих рішень щодо інших операторів. Спосіб виведення *x* залежить від вибору структури даних, тому спочатку деталізуємо обчислення степеня:

```
void power2()
{
    int n;
    cout << "Введіть n: "; cin >> n;
    велике_число x;
    // обчислити 2^n в змінній x
    x = велике_1;
    for (int k = 1; k <= n; ++k)
    {
        подвоїти_велике_число(x);
    }
    cout << "2 в степені " << n << " = ";
    надрукувати_велике_число(x);
}
```

У цьому випадку для обчислення 2^n вибрано послідовне множення, а не, наприклад, швидке піднесення до степеня. Для деталізації використано дві структури керування: послідовність і цикл.

Подальша деталізація можлива лише після вибору структури даних для реалізації типу *велике_число*. Такий тип можна було б змоделювати за допомогою масиву або лінійного однозв'язного списку. Залежно від розміру елемента масиву (чи списку) його використовують з метою зберігання різної кількості цифр великого числа: якщо розмір байт (тип *char*), то один елемент масиву містить одну або дві цифри числа; якщо два байти (тип *short int*), то один елемент може містити до чотирьох цифр числа; якщо чотири байти (тип *int*), – до дев'яти цифр. Зупинимо свій вибір на найпростішому варіанті і використаємо масив байтів, кожен з яких призначено для зберігання однієї цифри великого числа. У цьому випадку визначають максимальну довжину масиву і використовують додаткову змінну для зберігання номера останнього зайнятого елемента масиву. Легко обчислити кількість цифр у числі 2^{1000}: $[\lg 2^{1000}]+1=302$. Тепер тип *велике_число* можна конкретизувати:

```
struct LargeNumber
{
    static const int length = 302;
    char digits[length];          // масив цифр
    int last;                     // номер останньої цифри
};

LargeNumber x;
```

Для зручності обчислень масив *x.digits* міститиме цифри числа *x* у зворотному порядку: наймолодша цифра записана в перший елемент масиву. Нехай вас не введе в оману використання символьного типу *char*. У мові C++ – це один з цілих типів, який можна використовувати для обчислень. Елементи масиву *digits* зберігатимуть числа від 0 до 9, а не літери від '0' до '9'.

Тепер можна конкретизувати інші абстрактні оператори з програми *power2*.

Замість *x* = *велике*_1;

```
x.last = 0;
x.digits[0] = 1;
```

Щоб уточнити *подвоїти_велике_число*(*x*), пригадаємо алгоритм множення числа на 2 у стовпчик: необхідно помножити кожну цифру на 2, додати до неї перенесення з попереднього розряду, записати число одиниць отриманої суми і перенести число її десятків до наступного розряду. Для запам'ятовування проміжних результатів (суми і перенесення) використаємо додаткові змінні *calc* і *transfer*. Зауважимо також, що перенесення не додають до наймолодшої цифри числа, але цю особливість легко подолати, поклавши перед початком множення *transfer* = 0. Остаточно замість *подвоїти_велике_число*(*x*)

```
    char calc;                        // результат проміжних обчислень
    char transfer = 0;                // перенесення до старшого розряду
    for (int i = 0; i <= x.last; ++i) // переглядаємо всі цифри великого числа
    {
        calc = x.digits[i] * 2 + transfer;
        x.digits[i] = calc % 10;      // молодшу цифру добутку записуємо
        transfer = calc / 10;         // старшу - переносимо
    }
    if (transfer > 0)                 // запис числа стає довшим
        x.digits[++x.last] = transfer;
```

Оператор *надрукувати_велике_число*(*x*) є звичайною послідовною обробкою елементів масиву. Єдиною особливістю є те, що першою в масиві стоїть остання цифра шуканого числа, тому друк розпочинаємо з останнього елемента масиву. Отже, замість *надрукувати_велике_число*(*x*):

```
    for (int i = x.last; i >= 0; --i) cout << (int)x.digits[i];
    cout << '\n';
```

Використання символьного типу таки далося взнаки. Для того, щоб правильно вивести на друк цифри великого числа, довелося повідомити компіляторові, що потрібно друкувати числа, а не літери, і застосувати приведення типу.

Ми записали мовою C++ усі абстрактні оператори і оголошення. Отже, деталізацію завершено. Тепер можна просто зібрати все в одну програму. Проте краще оформити «мудрі» оператори у вигляді процедур. Адже процедура – це і є великий («мудрий») оператор:

```
struct LargeNumber
{
    static const int length = 302;
    char digits[length];          // масив цифр
    int last;                     // номер останньої цифри
};
```

```
// процедура подвоєння великого числа
void MultByTwo(LargeNumber& x)
{
    char calc;                          // результат проміжних обчислень
    char transfer = 0;                  // перенесення до старшого розряду
    for (int i = 0; i <= x.last; ++i) // переглядаємо всі цифри великого числа
    {
        calc = x.digits[i] * 2 + transfer;
        x.digits[i] = calc % 10;        // молодшу цифру добутку записуємо
        transfer = calc / 10;           // старшу - переносимо
    }
    if (transfer > 0)                   // запис числа стає довшим
        x.digits[++x.last] = transfer;
}

// процедура виведення великого числа
void Print(const LargeNumber& x)
{
    for (int i = x.last; i >= 0; --i) cout << (int)x.digits[i];
    cout << '\n';
}

// програма піднесення 2 до степеня n, 1<=n<=1000
void PowerLast()
{
    int n;
    cout << "Введіть n (n<=1000): "; cin >> n;
    LargeNumber x; // велике число x

    // обчислити 2^n в змінній x
    //     x = велике_1;
    x.last = 0;
    x.digits[0] = 1;  // x = 2^0

    for (int k = 1; k <= n; ++k)
    {
        // подвоїти_велике_число(x);
        MultByTwo(x); // x = 2^k
    }
    // x = 2^n
    cout << "2 в степені " << n << " = ";
    //надрукувати_велике_число(x);
    Print(x);
}
```

Обчислення великого степеня числа 2 можна виконати по-іншому, якщо вибрати швидкий алгоритм піднесення до степеня, або якщо обрати іншу структуру даних для зберігання великого числа. З метою порівняння наведемо ще одну програму обчислення 2^n, у якій використано однозв'язний список. У цьому випадку немає практично жодних обмежень на величину показника *n*. Таке обмеження ми раніше використовували для обчислення найбільшого розміру масиву. А розмір списку може змінюватися динамічно: за необхідністю до нього долучають нові ланки для запису нових цифр числа. Головна відмінність нової програми – процедура очищення динамічної пам'яті від ланок списку. Зміниться також логіка виведення великого числа. Оскільки однозв'язний список можна перебирати лише від першої ланки до останньої, а наш список розпочинатиметься з

наймолодшої цифри, то перед виведенням його доведеться обернути (або створити обернуту копію).

Для елементів списку використаємо тип *unsigned short*, що дасть змогу записувати в них від одної до чотирьох цифр великого числа і зменшити удвічі порівняно з попереднім розв'язком кількість елементів структури даних. Для того, щоб така «узагальнена цифра» правильно друкувалася, наприклад, щоб число 17 було надруковане у вигляді '0017', налаштуємо відповідно потік виведення.

```
// спискова структура для зображення великого числа
struct NumNode
{
	unsigned short num;
	NumNode* link;
	NumNode(unsigned short x, NumNode* p = nullptr) :num(x), link(p) {}
};

// процедура звільнення динамічної пам'яті
void EraseList(NumNode*& x)
{
	while (x != nullptr)
	{
		NumNode* victim = x;
		x = x->link;
		delete victim;
	}
}

// процедура подвоєння великого числа
void MultByTwo(NumNode* x)
{
	unsigned short calc;
	unsigned short transfer = 0;
	// заголовна ланка потрібна для правильного адресування
	x = new NumNode(0, x);
	NumNode* curr = x;			// допоміжний вказівник для перебирання ланок
	while (curr->link != nullptr)
	{
		calc = curr->link->num * 2 + transfer;
		if (calc < 10000)
		{				// всі 4 цифри поміщаються в одній ланці
			transfer = 0;
			curr->link->num = calc;
		}
		else
		{				// є перенесення в наступну ланку
			transfer = 1;
			curr->link->num = calc % 10000;
		}
		curr = curr->link;
	}
	// Нова ланка, якщо число стало довшим
	if (transfer > 0) curr->link = new NumNode(transfer);
	// Заголовна ланка більше не потібна
	delete x;
}
```

```
// процедура виведення великого числа
void Print(const NumNode* x)
{
	// Перед виведенням список треба обернути
	NumNode* reverse = nullptr;
	while (x != nullptr)
	{
		reverse = new NumNode(x->num, reverse);
		x = x->link;
	}
	x = reverse;
	cout << x->num;	// друк старшої "цифри" без ведучих нулів
	x = x->link;

	char empt = cout.fill('0');
	while (x != nullptr) // решту "цифр" друкуємо по чотири і з нулями
	{
		cout.width(4);
		cout << x->num;
		x = x->link;
	}
	cout << '\n';

	cout.fill(empt); // повертаємо заповнювач за замовчуванням
	EraseList(reverse); // обернутий список більше не потрібен
}

// програма піднесення 2 до степеня n
void PowerByList()
{
	cout << "* Обчислення великого степеня числа 2 *\n\n";
	int n;
	cout << "Введіть n: "; cin >> n;
	NumNode* x; // велике число x

	// обчислити 2^n в змінній x
	//	x = велике_1;
	x = new NumNode(1);  // x = 2^0

	for (int k = 1; k <= n; ++k)
	{
		// подвоїти_велике_число(x);
		MultByTwo(x); // x = 2^k
	}
	// x = 2^n
	cout << "2 в степені " << n << " = ";
	//надрукувати_велике_число(x);
	Print(x);

	EraseList(x);
}
```

Ми свідомо не деталізуємо опис процесу створення програми *PowerByList*. Структуровану програму легко модифікувати, пристосовуючи до нових умов (до використання іншої структури даних у нашому випадку). Читач має змогу самостійно

порівняти ці дві програми, зазначити їхні особливості, зумовлені відмінністю використаних структур даних і різними типами елементів цих структур.

Обидві попередні програми написано в процедурному стилі, проте мова C++ надає набагато зручніші засоби програмування: ми могли б оголосити клас «велике число» і перевантажити для нього потрібні оператори, так програма матиме зрозуміліший вигляд. Для зберігання цифр числа можемо використати стандартний контейнер бібліотеки STL. Так нам не доведеться турбуватися про розподіл пам'яті для послідовності цифр.

```
// клас для зображення великого числа
class LargePositiveInteger
{
private:
	vector<char> m_digits;
public:
	LargePositiveInteger() { m_digits.push_back(0); }
	LargePositiveInteger& operator=(unsigned);
	LargePositiveInteger& MultByTwo();
	LargePositiveInteger& operator*=(unsigned);
	friend ostream& operator<<(ostream&, const LargePositiveInteger&);
};

LargePositiveInteger& LargePositiveInteger::operator=(unsigned n)
{
	m_digits.clear();
	if (n == 0) m_digits.push_back(0);
	else
	{
		while (n > 0)
		{
			m_digits.push_back(n % 10);
			n /= 10;
		}
	}
	return *this;
}

LargePositiveInteger& LargePositiveInteger::MultByTwo()
{
	char calc;
	char transfer = 0;
	for (int i = 0; i < m_digits.size(); ++i)
	{
		calc = m_digits[i] * 2 + transfer;
		m_digits[i] = calc % 10;
		transfer = calc / 10;
	}
	if (transfer > 0) m_digits.push_back(transfer);
	return *this;
}

LargePositiveInteger& LargePositiveInteger::operator*=(unsigned n)
{
	if (n == 0) return *this = 0;
	unsigned calc;
	unsigned transfer = 0;
	for (int i = 0; i < m_digits.size(); ++i)
```

```
	{
		calc = m_digits[i] * n + transfer;
		m_digits[i] = calc % 10;
		transfer = calc / 10;
	}
	while (transfer > 0)
	{
		m_digits.push_back(transfer % 10);
		transfer /= 10;
	}
	return *this;
}

ostream& operator<<(ostream& os, const LargePositiveInteger& x)
{
	for (int i = x. m_digits.size() - 1; i >= 0; --i)
		os << (int)x. m_digits[i];
	os << '\n';
	return os;
}

void PowerLastest()
{
	cout << "* Обчислення великого степеня числа 2 *\n\n";
	int n;
	cout << "Введіть n: "; cin >> n;
	LargePositiveInteger x; // велике число x
	// обчислити 2^n в змінній x
	//     x = велике_1;
	x = 1;  // x = 2^0

	for (int k = 1; k <= n; ++k)
	{
		// подвоїти_велике_число(x);
		x *= 2; // x = 2^k
		// x.MultByTwo();
	}
	// x == 2^n
	//надрукувати_велике_число(x);
	cout << "2 в степені " << n << " = " << x;
}
```

Цікаво, що клас *LargePositiveInteger* можна використати і для інших обчислень, наприклад, щоб отримати факторіал великого числа. Пропонуємо зробити це читачеві самостійно.

Описані алгоритми множення великих чисел не претендують на найвищу ефективність. Зрозуміло, що для повноцінних обчислень бракує реалізації решти арифметичних операторів. У багатьох мовах програмування подання цілих чисел необмеженої величини та дії з ними реалізовано на рівні компілятора. Такі засоби мають Lisp, Pharo, Python, середовища Maple, Mathematica. У мові C# можна використовувати спеціальний клас. У програмах мовою C++ також варто використати відповідну бібліотеку, наприклад, описану в https://www.codeproject.com/Articles/5319814/Arbitrary-Precision-Easy-to-use-Cplusplus-Library.

9.2. Запитання та завдання для самоперевірки

1. Сформулюйте ідею методу покрокової розробки програм. Що таке структурне програмування «зверху-донизу»?
2. Які частини абстрактної програми деталізують найперше при застосуванні методу покрокової розробки програм? Коли завершують деталізацію?
3. Назвіть два головні результати застосування деталізації абстрактної програми.
4. Спочатку в програмі для зберігання цифр результату було використано масив байтів. На які частини програми вплинув перехід до використання зв'язного списку? Як?
5. Функція виведення великого числа – списку *void Print(const NumNode* x)* спочатку обертає цей список, щоб отримати правильний порядок цифр. Поміркуйте, як за допомогою рекурсії обійтися без обертання списку.
6. Опишіть переваги об'єктно-орієнтованого підходу до розв'язування задачі обчислення великого степеня цілого числа.
7. З якою метою в типі *LargePositiveInteger* оголошено метод *operator*=* ?
8. Завантажте програми за наведеним посиланням, запустіть їх на виконання.
9. Використайте екземпляр типу *LargePositiveInteger* для обчислення факторіалу великого натурального числа, наприклад, 1000!.
10. Доповніть на власний розсуд можливості розробленого типу *LargePositiveInteger*.
11. Випробуйте бібліотеку Arbitrary Precision для роботи з довгими цілими, описану на codeproject.com (посилання наприкінці параграфа).